In [1]:
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import interpolate
from mpl_toolkits.mplot3d import Axes3D
from collections import Counter
import numpy as np
from scipy.spatial import cKDTree
```

Question 1

In [5]:
```python
# Differential Equation
def diffunc(x,y):
    return 1/(x**2 * (1-y))
```

Question 1.b

In [6]:
```python
# Euler Method
def euler(x0, y, h, x):
    # Iterating till x
    while x0 > x:
        y = y + h * diffunc(x0, y)
        x0 = x0 + h
    print("Approximate solution at x = ", x, " is ", "%.6f"% y)

euler(1, -1, -0.05, 0)
euler(1, -1, -0.05, 0.05)
```

```
Approximate solution at x =  0  is  -8.124934
Approximate solution at x =  0.05  is  -4.465853
```

Question 1.C

In [10]:
```python
# 4th-order Runge-Kutta
def runge_kutta(x0, y, h, x):
    # Iterate til x
    while x0 > x:
        k_1 = h * diffunc(x0,y)
        k_2 = h * diffunc(x0 + (1/2) * h, y + (1/2) * k_1)
        k_3 = h * diffunc(x0 + (1/2) * h, y + (1/2) * k_2)
        k_4 = h * diffunc(x0 + h, y + k_3)
        y = y + (1/6) * k_1 + (1/3) * k_2 + (1/3) * k_3 + (1/6) * k_4
        x0 = x0 + h
    print("Approximate solution at x = ", x, " is ", "%.6f"% y)


runge_kutta(1, -1, -0.05, 0)
runge_kutta(1, -1, -0.05, 0.05)
```

```
Approximate solution at x =  0  is  -607173044131906172468854784 0.000000
Approximate solution at x =  0.05  is  -5.500434
```

Question 1.d

In [73]:
```
def inte(F,x,y,xStop,tol):
    def midpoint(F, x0, y0, xStop, nSteps):
        h = (xStop - x0) / nSteps
        x = x0
        y_prev = y0
        y_curr = y_prev + h * F(x, y_prev)
        for i in range(1, nSteps):
            x = x + h
            y_next = y_prev + 2.0 * h * F(x, y_curr)
            y_prev = y_curr
            y_curr = y_next
        x = x + h
        y_end = 0.5 * (y_curr + y_prev + h * F(x, y_curr))
        return y_end

    def richardson(r, nSteps_list):
        k = len(r) - 1
        for j in range(k, 0, -1):
            factor = (nSteps_list[k] / nSteps_list[j - 1]) ** 2 - 1
            r[j - 1] = r[j] + (r[j] - r[j - 1]) / factor
        return
    # Maximum iteration
    kMax = 8
    #nSteps_list = [2 ** k for k in range(1, kMax + 1)]
    # Substep size
    nSteps_list = [2*(k+1) for k in range(0, kMax)]
    r = []
    for nSteps in nSteps_list:
        y_mid = midpoint(F, x, y, xStop, nSteps)
        r.append(y_mid)
        if len(r) > 1:
            richardson(r, nSteps_list[:len(r)])
            error = abs(r[0] - r[1])
            if error < tol:
                return r[0]
    print("Warning: Maximum iterations reached without convergence.")
    return r[0]

def bulStoer(F, x, y, xStop, H, tol=1.0e-6):
    X = [x]
    Y = [y]
    while abs(x - xStop) > 1e-10:
        H_step = min(H, xStop - x) if x < xStop else max(H, xStop - x)
        y = inte(F, x, y, x + H_step, tol)
        x = x + H_step
        X.append(x)
        Y.append(y)
    return array(X), array(Y)
def F(x, y):
    return 1 / (x ** 2 * (1 - y))


x0 = 1
y0 = -1
xStop = 0.05
H = -0.05
tol = 1e-6
```

```
X, Y = bulStoer(F, x0, y0, xStop, H, tol)
print("Approximate solution at x =", xStop, "is", Y[-1])
```

Approximate solution at x = 0.05 is -5.48074058058637

Question 2

In [389…

```
def plot_3D(data):
    fig = plt.figure(figsize=(10, 8))
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(data[:, 0], data[:, 1], data[:, 2], c='blue', marker='o', s=10)
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    ax.set_title('3D Point Cloud Data')
    plt.show()

def plot_2D(data, title='Projection', label="Projected Points"):
    plt.figure(figsize=(8, 6))
    plt.scatter(data[:, 0], data[:, 1], color='blue', alpha=0.7, label=label)
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.title(title)
    plt.legend()
    plt.grid(True)
    plt.show()
```

Empty Table

In [391…

```
def PCA(data):
    # Normalize
    pc = data - np.mean(data, axis=0)
    # Covariance Matrix
    Q = np.dot(pc.T, pc) / (pc.shape[0] - 1)
    U, Sigma, Vt = np.linalg.svd(Q)
    # Eigenvectors & Eigenvalue
    eigenvectors = U
    eigenvalues = Sigma
    # Normal Vector of plane
    normal_vector = eigenvectors[:, np.argmin(eigenvalues)]
    # Principle Componenet
    largest_vector = eigenvectors[:, np.argmax(eigenvalues)]
    second_largest_vector = eigenvectors[:, np.argsort(eigenvalues)[-2]]
    principal_component = np.hstack((largest_vector[:, np.newaxis], second_large
    return normal_vector, principal_component


def PlaneParam(normal_vector,data):
    '''
    Given a normal vector and data of a plane, return
    the parameter of the plane
    '''
    centorid = np.mean(data, axis=0)
    a, b, c = normal_vector
    d = - (a * centorid[0] + b * centorid[1] + c * centorid[2])
    return a, b, c, d
```

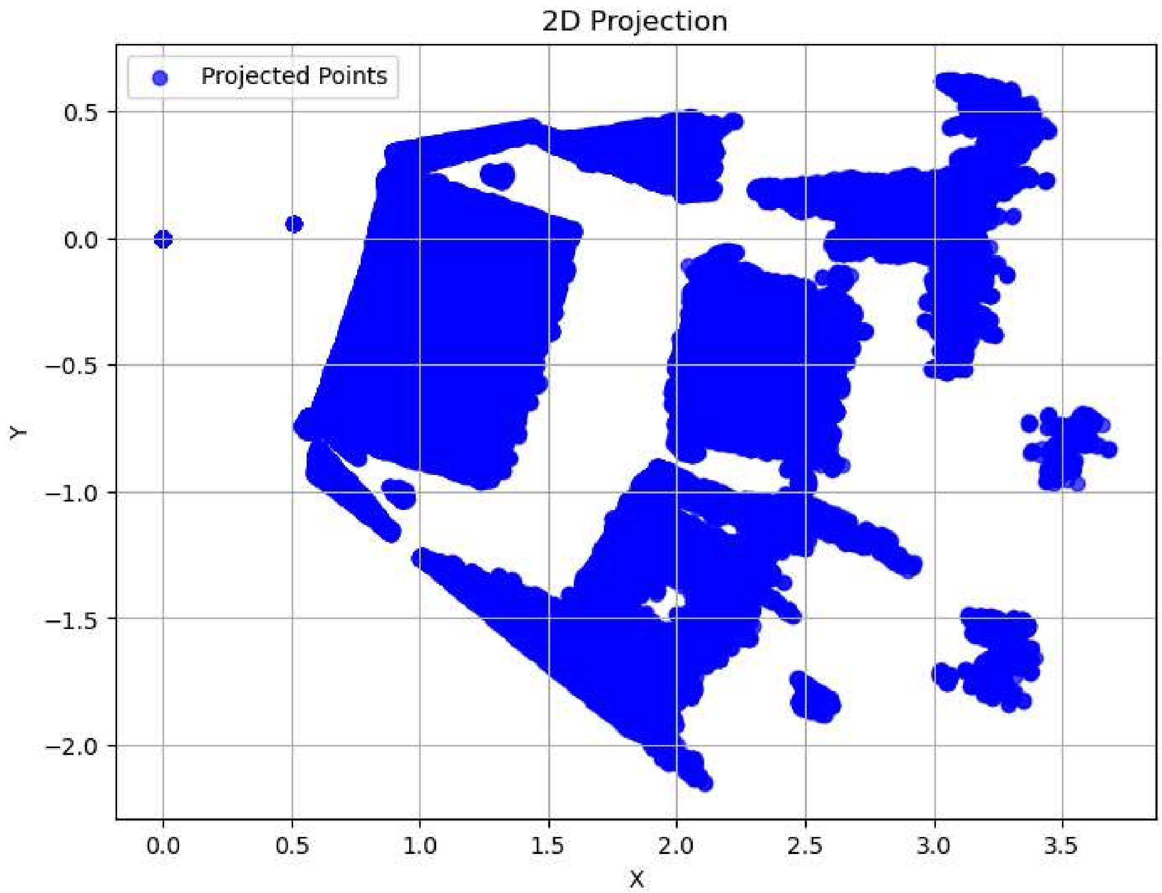
```python
def PlaneProject(principle_component, data):
    '''
    Given a principle component of data, make projection
    of data into the plane of principle component
    '''
    return np.dot(data, principal_components)


def planeFit(data):
    normal, principle = PCA(data)
    a,b,c,d = PlaneParam(normal, data)
    pc_project = PlaneProject(principle,data)
    print('Major Plane Parameter')
    print(f'a:{a}')
    print(f'b:{b}')
    print(f'c:{c}')
    print(f'd:{d}')
    plot_2D(pc_project)

def voxel_downsample(data, voxel_size=0.02):
    coords = np.floor(data / voxel_size).astype(int)
    unique_coords, indices = np.unique(coords, axis=0, return_index=True)
    return data[indices]

def statistical_outlier_removal(data, nb_neighbors=20, std_ratio=2.0):
    tree = cKDTree(data)
    distances, _ = tree.query(data, k=nb_neighbors)
    mean_distances = np.mean(distances, axis=1)
    threshold = np.mean(mean_distances) + std_ratio * np.std(mean_distances)
    mask = mean_distances < threshold
    return data[mask]
```

Below are my first attempt, just perform PCA and plane fitting on dominant plane

In [344…

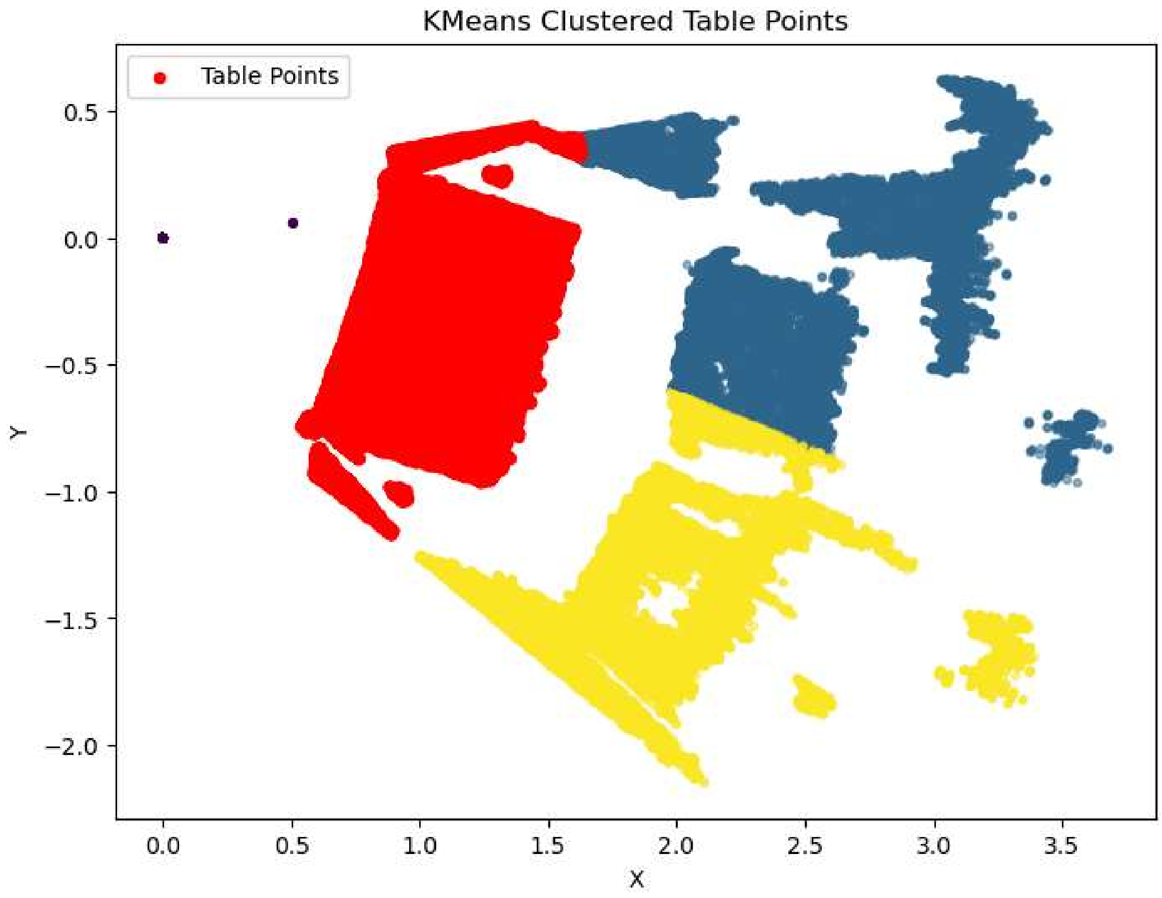
```python
pc_raw = np.loadtxt('./Empty2-1.asc')
normal, principle = PCA(pc_raw)
a,b,c,d = PlaneParam(normal, pc_raw)
pc_project = PlaneProject(principle,pc_raw)
print('Major Plane Parameter')
print(f'a:{a}')
print(f'b:{b}')
print(f'c:{c}')
print(f'd:{d}')
plot_2D(pc_project)
```

```
Major Plane Parameter
a:0.08751468415007803
b:0.9631746638292203
c:0.2542356131929213
d:-0.3031026814273146
```



In [346…

```
from sklearn.cluster import KMeans
n_clusters = 4
kmeans = KMeans(n_clusters=n_clusters, random_state=0)

# I tried to use KMEAN to find table
labels = kmeans.fit_predict(pc_project)
label_counts = Counter(labels)
table_label = label_counts.most_common(1)[0][0]
table_points = pc_project[labels == table_label]
plt.figure(figsize=(8, 6))
plt.scatter(pc_project[:, 0], pc_project[:, 1], c=labels, cmap='viridis', marker
plt.scatter(table_points[:, 0], table_points[:, 1], color='red', marker='o', s=2
plt.xlabel("X")
plt.ylabel("Y")
plt.title("KMeans Clustered Table Points")
plt.legend()
plt.show()
```



Below is a modified method

- Use RANSAC to find fit multiple plane from the point cloud, and only extract the inlier of plane with the most points
- Use DBSCAN clustering to find the table from the inlier
- Do PCA and make projection of table

In [404…

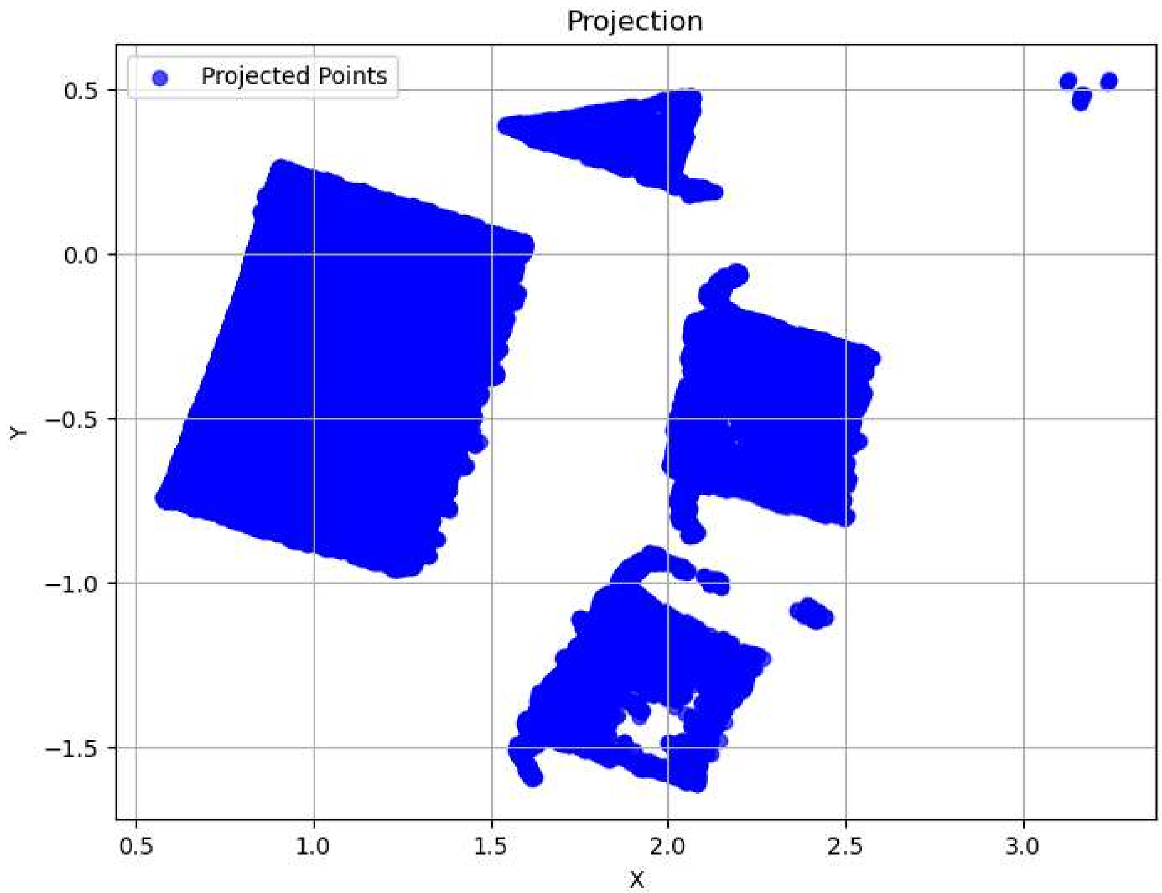
```
import numpy as np
from scipy.spatial import cKDTree
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn.linear_model import RANSACRegressor
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import DBSCAN

def findLargestPlane(data, min_points=50, residual=0.1):
    inliers_list = []
    min_points_for_plane = min_points

    while len(data) > min_points_for_plane:
        scaler = StandardScaler()
        data_scaled = scaler.fit_transform(data)

        # Plan fitting
        X = data_scaled[:, :2]
        y = data_scaled[:, 2]
        ransac = RANSACRegressor(residual_threshold=residual, max_trials=1000)
        ransac.fit(X, y)

        # Extract inlier for each plan
```

```
        inlier_mask = ransac.inlier_mask_
        if np.sum(inlier_mask) < min_points_for_plane:
            break
        inliers = data[inlier_mask]
        inliers_list.append(inliers)
        data = data[~inlier_mask]
    return inliers_list[0]


def findLargestCluster(inlier, eps=0.05, min_point=10):
    inlier_scaled = StandardScaler().fit_transform(inlier)
    # Apply DBSCAN clustering
    dbscan = DBSCAN(eps=0.05, min_samples=10)
    labels = dbscan.fit_predict(inlier_scaled)

    # Count the number of points in each cluster
    unique_labels, counts = np.unique(labels, return_counts=True)
    largest_cluster_label = unique_labels[np.argmax(counts)]
    largest_cluster_points = inlier[labels == largest_cluster_label]
    return largest_cluster_points
```

In [393…

```
data = np.loadtxt('./Empty2-1.asc')
plane = findLargestPlane(data, 50, 0.1)
planeFit(plane)
```

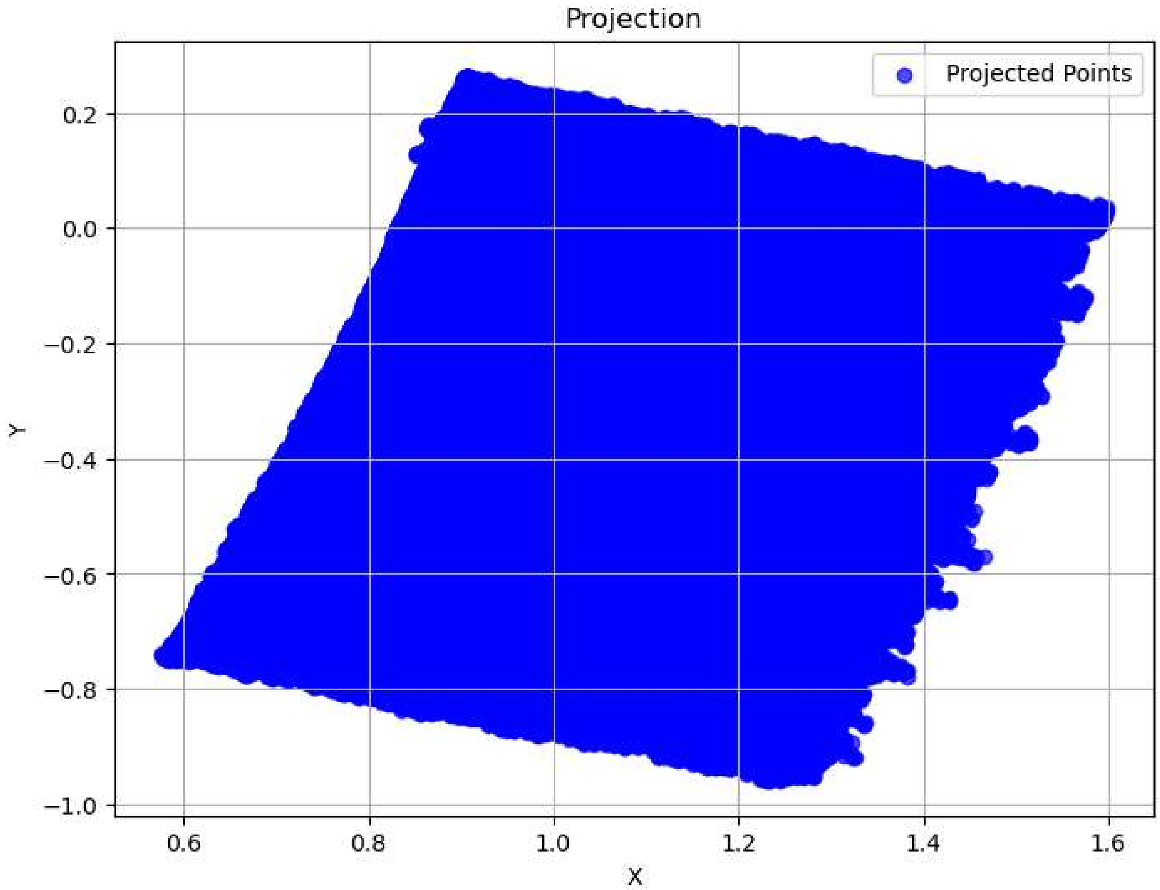
```
Major Plane Parameter
a:0.002672332282294145
b:0.9115026590987191
c:0.41128549828086247
d:-0.5785771337209781
```



In [399…

```
table = findLargestCluster(plane, 0.05, 10)
planeFit(table)
```

```
Major Plane Parameter
a:0.0007352700619255972
b:0.9034278478956234
c:0.4287395281807107
d:-0.5995175646371099
```



Clustered Tabe

I Switched to MATLAB for the rest questions, I put my MATLAB codes in the cell to make PDF

In [ ]:
```
% Read Data
data = readmatrix('TableWithObjects2-1.asc');
ptCloud = pcdenoise(pointCloud(data));
pointData = ptCloud.Location;

% Parameters
% Initial Distance & Angle Threshold
initialDistanceThreshold = 0.2;
initialAngleThreshold = 50;
% Minimum Distance & Angle Threshold
minDistanceThreshold = 0.052;
minAngleThreshold = 5;
% Max times of iteration
maxIterations = 10;
tolerance = 1e-3;


currentDistanceThreshold = initialDistanceThreshold;
currentAngleThreshold = initialAngleThreshold;
```

```
for i = 1:maxIterations
    % PCA: Compute Dominate Plane
    [coeff, ~, ~, ~, ~, mu] = pca(pointData);
    normal_vector = coeff(:, 3);

    % Distance of each point to Dominate Plane
    point_cloud_centered = pointData - mu;
    distances = abs(point_cloud_centered * normal_vector);

    % Angle Difference of each point to Dominate Plane
    normals = pcnormals(pointCloud(pointData), 10);
    angleDifferences = acosd(dot(normals, repmat(normal_vector', size(normals, 1

    % Filterout outliers
    inliers = pointData((distances < currentDistanceThreshold) & (angleDifferenc

    % Check Convergence
    if size(inliers, 1) / size(pointData, 1) > (1 - tolerance)
        break;
    end
    pointData = inliers;

    % Reduce Angle & Distance Threshold
    currentDistanceThreshold = max(currentDistanceThreshold * 0.8, minDistanceTh
    currentAngleThreshold = max(currentAngleThreshold * 0.8, minAngleThreshold);
end


% Cluster remining inliers
epsilon = 0.3;
minPts = 10;
labels = dbscan(inliers, epsilon, minPts);
uniqueLabels = unique(labels);
% Table should be the clusters with the most dense points
maxClusterLabel = mode(labels(labels >= 0));
inliers = inliers(labels == maxClusterLabel, :);

% Plot point cloud
figure;
scatter3(inliers(:, 1), inliers(:, 2), inliers(:, 3), 10, inliers(:, 3), 'fillec
xlabel('X');
ylabel('Y');
zlabel('Z');
title(['Filtered Point Cloud with ' num2str(maxIterations) ' iterations']);
axis equal;
grid on;


% Compute Table inlier
[finalCoeff, ~, ~, ~, ~, finalMu] = pca(inliers);
planeNormal = finalCoeff(:, 3);
planePoint = finalMu;
a = planeNormal(1);
b = planeNormal(2);
c = planeNormal(3);
d = -dot(planeNormal, planePoint);
fprintf('Table Parameter: %fx + %fy + %fz + %f = 0\n', a, b, c, d);
```

CSE

In [ ]:
```
data = readmatrix('CSE-1.asc');
ptCloud = pcdenoise(pointCloud(data));
pointData = ptCloud.Location;

% Find Floor
% PCA: Compute Floor Plane
[coeff, ~, ~, ~, ~, mu] = pca(pointData);
normal_vector = coeff(:, 3);
maxDistance = 0.5;
referenceVector = normal_vector;
maxAngularDistance = 5;
% Plane Fitting: Floor Plane
[model1,inlierIndices,outlierIndices] = pcfitplane(ptCloud,...
    maxDistance,referenceVector,maxAngularDistance);
plane1 = select(ptCloud,inlierIndices);
remainPtCloud = select(ptCloud,outlierIndices);

% Find First Wall
% PCA: Compute First Wall
remaining_points = remainPtCloud.Location;
[wall_coeff, ~, ~, ~, ~, ~] = pca(remaining_points);
% First Wall Normal: Perpendicular to floor
angles = zeros(3,1);
floor_normal = normal_vector;
for i = 1:3
    angles(i) = abs(dot(wall_coeff(:,i), floor_normal));
end
[~, idx] = min(angles);
wall_reference = wall_coeff(:, idx);
wall_reference = wall_reference - (dot(wall_reference, floor_normal) * floor_nor
wall_reference = wall_reference / norm(wall_reference);
% Plane Fitting: First Wall
maxDistance = 0.6;
maxAngularDistance = 10;
[wallModel, wallInliers, wallOutliers] = pcfitplane(remainPtCloud, ...
    maxDistance, wall_reference, maxAngularDistance);
wall1 = select(remainPtCloud, wallInliers);

% Find Remaining Wall and Objects
remaining_points = select(remainPtCloud, wallOutliers);
wall_planes = {};
minPoints = 500;
wall_plane_param = {};
for iter = 1:5
    maxDistance = 0.5;
    maxAngularDistance = 10;
    % Wall & Object Normal: Perpendicular to First Wall and Floor
    referenceVector = cross(floor_normal, first_wall_normal);
    referenceVector = referenceVector / norm(referenceVector);
    % Plane Fitting
    [model, inlierIndices, outlierIndices] = pcfitplane(remaining_points, ...
        maxDistance, referenceVector, maxAngularDistance);
    if numel(inlierIndices) < minPoints
        break;
    end
    new_wall = select(remaining_points, inlierIndices);
```

```
    wall_planes{end+1} = new_wall;
    wall_plane_param{end+1} = model;

    % Remove found Wall & Object From the cloud
    validOutliers = outlierIndices(outlierIndices <= remaining_points.Count);
    remaining_points = select(remaining_points, validOutliers);
end
% Visualize
figure;
% Floor
pcshow(plane1.Location, 'g');
hold on;
% First Wall
pcshow(wall1.Location, 'r');
fprintf('Parameters of red wall:\n');
disp(wallModel);
% Each other Objects & Wall will be assigned with random color
for i = 1:length(wall_planes)
    pcshow(wall_planes{i}.Location, rand(1, 3));
    wall_center = mean(wall_planes{i}.Location, 1);
    text(wall_center(1), wall_center(2), wall_center(3), sprintf('Wall %d', i),
         'Color', 'r', 'FontSize', 12, 'FontWeight', 'bold');
    fprintf('Parameters of wall %d:\n', i);
    disp(wall_plane_param{i});
end
% Remaining Point Will be Black
pcshow(remaining_points.Location, 'k');
title('Ground and Walls');
xlabel('X');
ylabel('Y');
zlabel('Z');
legend('Floor', 'Wall 1', 'Additional Walls', 'Remaining Points');
```